1995

A Parallel Algorithm for Computing Minimum Spanning Trees

Donald B. Johnson
Dartmouth College

P. Takis Metaxas
Wellesley College, pmetaxas@wellesley.edu

A Parallel Algorithm for Computing Minimum Spanning Trees

Donald B. Johnson*
Dartmouth College‡

Panagiotis Metaxas†
Wellesley College§

*Email address: djohnson@dartmouth.edu, telephone: (603) 646-3385

†Part of this work was done while this author was with the Mathematics and Computer Science department, Dartmouth College. Email address: pmetaxas@wellesley.edu, telephone: (617) 283-3054

‡6211 Sudikoff Laboratory for Computer Science, Hanover, NH 03755

§Department of Computer Science, Wellesley, MA 02181-8289

Running Head: Parallel Minimum Spanning Tree Algorithm.

Keywords: Parallel Algorithms, Graph Algorithms, Minimum Spanning Tree, EREW PRAM model.

Corresponding Address: Panagiotis T. Metaxas, Department of Computer Science, Wellesley College, Wellesley, MA 02181-8289.

Abstract

We present a simple and implementable algorithm that computes a minimum spanning tree of an undirected weighted graph $G = (V, E)$ of $n = |V|$ vertices and $m = |E|$ edges on an EREW PRAM in $O(\log^{3/2} n)$ time using $n+m$ processors. This represents a substantial improvement in the running time over the previous results for this problem using at the same time the weakest of the PRAM models. It also implies the existence of algorithms having the same complexity bounds for the EREW PRAM, for connectivity, ear decomposition, biconnectivity, strong orientation, st-numbering and Euler tours problems.

List of Symbols

$O()$ Capital Oh, slanted (math)
O Capital Oh
0 zero
l ell
1 one
$o()$ Lowercase oh, slanted (math)
o lowercase oh
α greek alpha

1 Introduction

This paper describes a new parallel algorithm for computing the minimum spanning tree (MST) of a graph in the EREW PRAM model of parallel computation, the weakest of the PRAM models. This algorithm is faster by a factor of $\sqrt{\log |V|}$ than any deterministic algorithm previously known for any model that does not make use of concurrent writing. The algorithm uses the growth-control scheduling of the connectivity algorithm described in [JM91]; it also makes use of an observation by [GGS89].

A major innovation is our discovery that necessary information about components can be extracted without ever explicitly shrinking the components. Component shrinking is a feature of every other parallel MST and connectivity algorithm known to us.

Two of our objectives while designing the algorithm were simplicity and implementability, that is, to be able to implement the algorithm using simple, well-understood routines (like sorting and list ranking) that are likely to be found on most parallel machines. We feel that we have succeeded in both. In fact, the complexity of our solution is in the proof — not in the algorithm itself.

Even though the connectivity algorithm of [JM91] improved the running time of several other graph-theoretic problems it seemed that there was no obvious way to create a MST algorithm from the connectivity algorithm having comparable complexity with the latter. The difficulty, of course, is that the selection of minimum weight edges from edge-lists seems to require either a powerful concurrent-write model of computation or some other minimization process, which thereby takes time logarithmic in the length of the list. A connectivity algorithm may select any edge, not the one with minimum weight, and that makes the selection simpler. Thus, a new approach was needed to achieve an $o(\log^2 |V|)$ running time for this problem. As we will explain, we maintain a subset of edges that contains all the edges that must be considered in any one phase of the algorithm in order to control the number of candidates that must be tested. Maintaining this subset is essential to the bound on the running time.

Our results. We present an algorithm that computes a minimum spanning tree (MST) of an undirected weighted graph $G = (V, E)$ of $n = |V|$ vertices and $m = |E|$ edges on an EREW PRAM in $O(\log^{3/2} n)$ time using $n + m$ processors. (If G is not connected, our algorithm finds a minimum spanning tree for each connected component.) This represents a substantial improvement in the running time over the previous results for this problem using at the same time the weakest of the PRAM models. It also implies the existence of a connectivity algorithm with the same complexity bounds for the EREW PRAM, therefore improving on previous work [JM91]. Furthermore, we note that the number of processors used can be reduced by a factor

of $O(\sqrt{\log n})$, provided that there exists a practical integer-sorting subroutine which runs in $O(\log n)$ time using $n/\sqrt{\log n}$ EREW PRAM processors. In this paper, we have not only succeeded in solving a problem more difficult than the connectivity problem (implying a new, simpler solution to the connectivity and related problems as well), but also have done so using the weakest of the PRAM models. We note that among the problems having running times depending on the connectivity algorithm are ear decomposition [MR86], biconnectivity [TV85], strong orientation [Vis85], *st*-numbering [MSV86] and Euler tours [AV84].

Computing the MST of a weighted graph has attracted much attention in both the sequential and parallel settings. The best known sequential algorithm runs in time $O(n^2)$ for dense graphs [Pri57], and in time $O(m \log_2 \log_2 \log_d n)$ for sparse graphs [GGS89], where $d = \max(m/n, 2)$. For a presentation of several sequential MST algorithms, see [Tar83, Chapter 6].

In parallel models, the previous results for the MST problem were $O(\log^2 n)$ using $n^2/\log^2 n$ CREW PRAM [HCS79, CLC82] or n^2 EREW PRAM processors [NM82], and $O(\log n)$ time using $n+m$ PRIORITY CRCW PRAM processors [AS87, SV82], or $(n+m)\log\log\log n/\log n$ STRONG CRCW PRAM processors [CV86] using very elaborate techniques. Other parallel algorithms are reported in [KRS90, KR84, Ben80, SJ81].

Recently, [CL93] have improved the running time of [JM91] to $O(\log n \log\log n)$ mainly by providing a recursive version of the growth-control schedule. It does not appear, however, that this technique has immediate application on the MST algorithm we present here.

The paper is organized as follows: Section 2 contains some preliminaries. Section 3 gives an outline of the algorithm and then describes its parts in some detail. Section 4 has the main theorem along with the correctness and complexity proofs. Finally, Section 5 contains the conclusions.

2 Preliminaries

2.1 Definitions

We give here some definitions, and we discuss the complexity of an algorithm that we use as a subroutine. The *minimum spanning tree* (MST) problem is defined as follows: Given a connected undirected graph $G = (V, E)$ each of whose edges has a real-valued *weight*, find a spanning tree of the graph whose total edge weight is minimum. A *pseudotree* $P = (C, D)$ is a maximal connected directed graph with $n = |C|$ vertices and $n = |D|$ arcs, for which each vertex has outdegree one. Every pseudotree has exactly one simple directed cycle. We call the number of arcs in the

cycle of a pseudotree P its *circumference*, $circ(P)$. A *rooted tree* is a pseudotree whose cycle is a loop on some vertex r called the *root*. A *rooted star* R with root r, is a rooted tree all of whose arcs point to r.

Given n elements in a linked list representation, the *list ranking* problem is to find, for each element, its distance from the end of the list, called its rank. The list ranking problem, which appears very often in parallel computation and will be used by our algorithm as well, can be solved optimally in $O(\log n)$ time using $n/\log n$ EREW PRAM processors [CV88, AM91].

Given a connected subgraph $G_i = (V_i, E_i) \subset G = (V, E)$, we define an *internal* edge to be an edge $(v, w) \in E_i$ such that $v, w \in V_i$. Similarly, we define an *outgoing* edge to be an edge $(v, w) \in E - E_i$ such that one of its endpoints belong to V_i and the other belongs to $V - V_i$. Let $G_j = (V_j, E_j)$ be another subgraph of G where G_i and G_j are vertex disjoint. Distinct edges $(v, w), (x, y) \in E - (E_i \cup E_j)$ having one endpoint in V_i and the other in V_j are called *multiple*.

Let $G = (V, E)$ be a connected weighted graph on $n = |V|$ vertices and $m = |E|$ edges, and let $weight : E \rightarrow R$ be a function which gives the weights of the m edges. We assume that the vertices of the graph are given in an array representation, and let $id(v)$ be the index of vertex v in the array. Each vertex v has a linked list $L(v)$ of edges (v, w) incident to vertex v and two pointers $first$ and $last$ pointing to the beginning and the end of $L(v)$. For implementation purposes we will assume that the last edge in every edge-list is a dummy one. There are two copies for each edge, $(v, w) \in L(v)$ and $(w, v) \in L(w)$, which are connected via a pair of $twin$ pointers. Finally, pointer $next(v, w)$ points to the next edge in (v, w)'s edge list.

2.2 The model

We briefly describe here the model of parallel computation we use. A PRAM (Parallel Random Access Machine) employs p processors, each one able to perform the usual computation of a sequential machine using some fixed amount of local memory. The processors communicate through a shared global memory to which all are connected. Depending on the way the access of the processors to the global memory is handled, PRAMs are classified as EREW, CREW and CRCW. (In the model names, E stands for "exclusive" and C for "concurrent".) If we don't allow any conflicts in the reading from and writing to the shared memory, the model is called an EREW PRAM, the weakest of the three models. If we allow only concurrent reading, we have a CREW PRAM. Finally, in the CRCW PRAM, simultaneous writing is permitted and we have to address the question of which of the attempting writing processors will write. In the PRIORITY CRCW PRAM model, the processor having the largest priority (id number) wins, while in the STRONG model the processor holding the minimum (or equivalently the maximum) of all the values attempted to be written wins.

One can simulate an algorithm designed for the PRIORITY CRCW model on a EREW PRAM, with a slowdown in time logarithmic in the number of processors used by the former machine [Eck77, Vis83].

It should be noted that for an algorithm to run on a model that permits no concurrent reads (writes), implies that any two processors make *no attempt* to concurrently read from (write to) the same memory location. If such an attempt is ever made, the result of the computation is undefined. For more information on the PRAM models, see [KR90]

3 Description of the Algorithm

3.1 Outline

The algorithm is divided into *phases* and maintains a minimum spanning forest of the graph. We will call each of the trees in the forest a *component*. Later on, when each component has grown in size by including sets of vertices and is organized as a rooted tree, the root will *represent* the component. In the beginning, we can think of each vertex as the root of the (trivial) component to which it belongs.

During each phase, each component C grows in size by executing the following two steps:

First, C finds the minimum-weight outgoing edge (v, w) which is connected to any of the vertices $v \in C$ and leads to vertex $w \in C'$ of some other component C'. This is called the *hooking* step (Figure 1). When executed simultaneously by all components, the hooking step creates clusters of components formed as pseudotrees with circumference 2 (asumming that all weights are distinct). Such pseudotrees will easily become rooted trees.

Each cluster produced by hooking is then processed into a new component C organized as a rooted tree with root r and one edge list $L(C)$. We call the completion of the creation of a new component *merging* (Figure 2). To merge the new component, a root r is chosen from the cycle with circumference two of the new component. Then, the edge list of r is augmented by all the other edge lists of the constituent components that hooked to form the new one. We use the EREW *edge-plugging scheme* [JM91, Met91] to perform the augmentation of r's edge-list in constant time and without memory access conflicts. Finally, housekeeping is performed on the merged edge list to remove internal and multiple edges.

One can easily see that repeating the sequence of the hooking and merging steps in parallel for a long enough period of time, a MST of the graph is computed.

Before we continue with the technical details, let us make an important observation. All previous algorithms for connectivity and MST, during the hooking step,

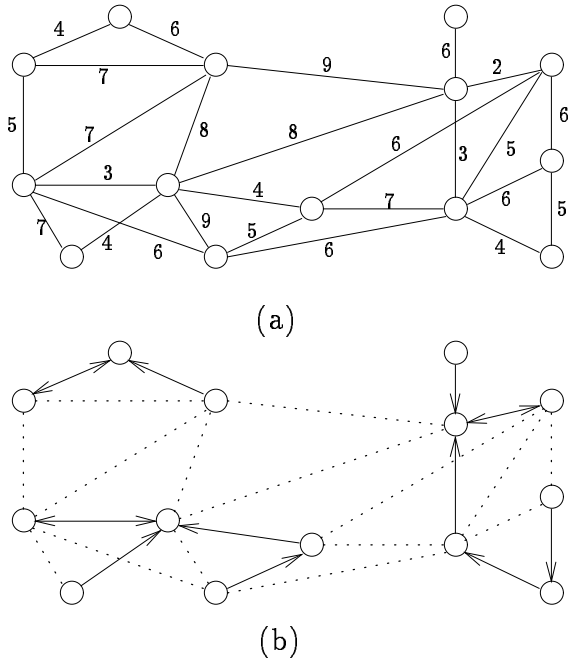

Figure 1: (a) The input graph G. Each vertex represents a component. (b) The hooking step: Each components has picked the minimum-weight outgoing edge. An arc points from a vertex to its selected neighboring component. Dotted are those edges that were not picked by any vertex. Three pseudotrees are shown in this figure. Note that each pseudotree contains a cycle of circumference two (shown as a double arc).

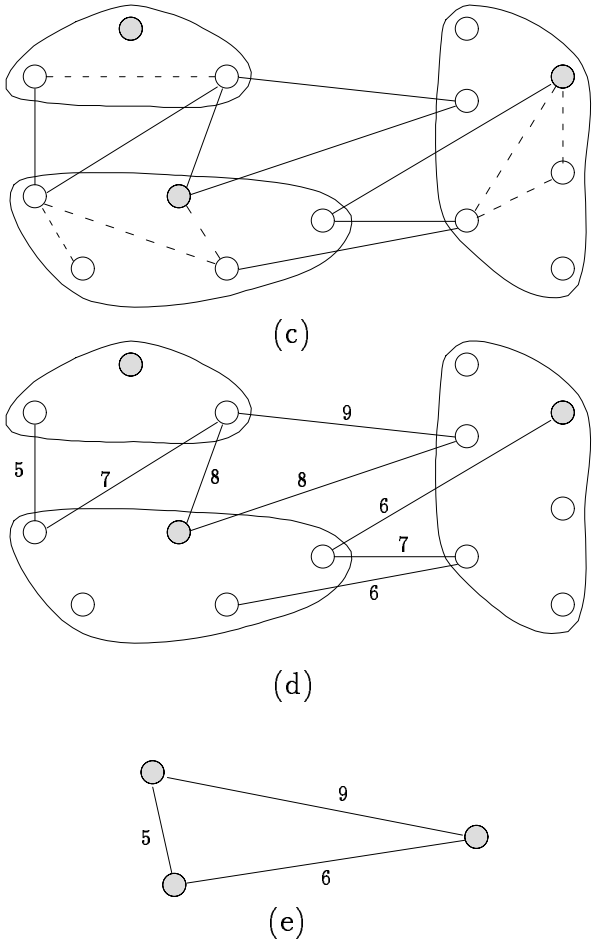

Figure 2: (c) The merging step. The new components have been identified. Shaded are the vertices that will become roots of the components/rooted trees. Dashed edges are internal edges that will be removed. (d) The new graph contains three components. Multiple edges are shown between the components. (e) The new graph after the removal of multiple edges.

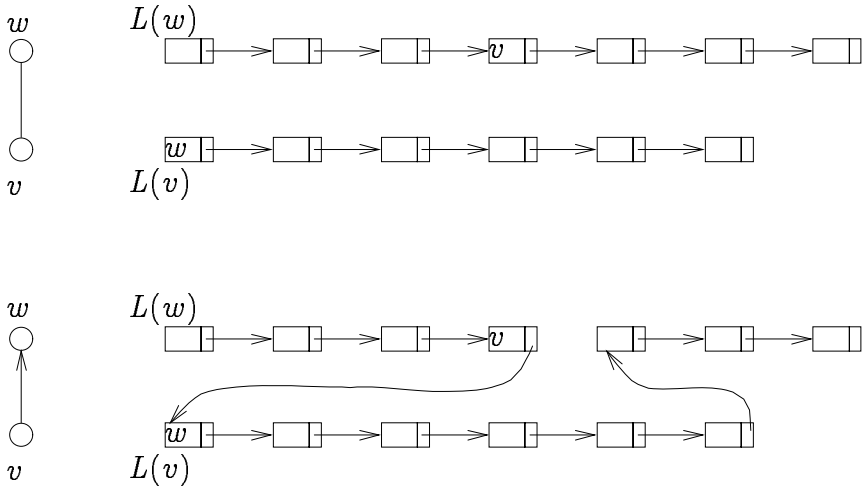

Figure 3: Edge lists of nodes v and w before (top) and after (bottom) the edge plugging step.

create trees or pseudotrees and then reduce them to explicit rooted stars by some kind of pointer jumping. Given that, in general, the in-degree of each node of the tree is not bounded by a constant, the reducing process generates read conflicts. In our algorithm we will avoid these conflicts, because we will never explicitly create these structures. Instead, for each new component C we will create a linked list $E(C)$, representing a preorder traversal of the component's tree. Then, we will use this linked list to gather the information about the component. Doing so obviates the need to shrink components.

Edge-Plugging. For reasons of completeness, we briefly describe here the edge-plugging scheme and how it ensures that no concurrent accesses happen when used.

As we mentioned in the Section 2.1, we represent each undirected edge (v, w) by two *twin* copies (v, w) and (w, v). The former is included in $L(v)$ and the latter in $L(w)$. The two copies are interconnected via a function $twin(e)$ which gives the address of the twin copy of edge e. We can assume that both (v, w) and (w, v) are being simulated by the same processor. Therefore, calculating the *twin* function in constant time is straightforward.

Let us assume that during the hooking step, edge (v, w) was chosen as the minimum-weight outgoing edge (Fig. 3, top). According to this scheme, v *plug*s its edge-list $L(v)$ into w's edge-list by redirecting some pointers. The exact place that $L(v)$ is

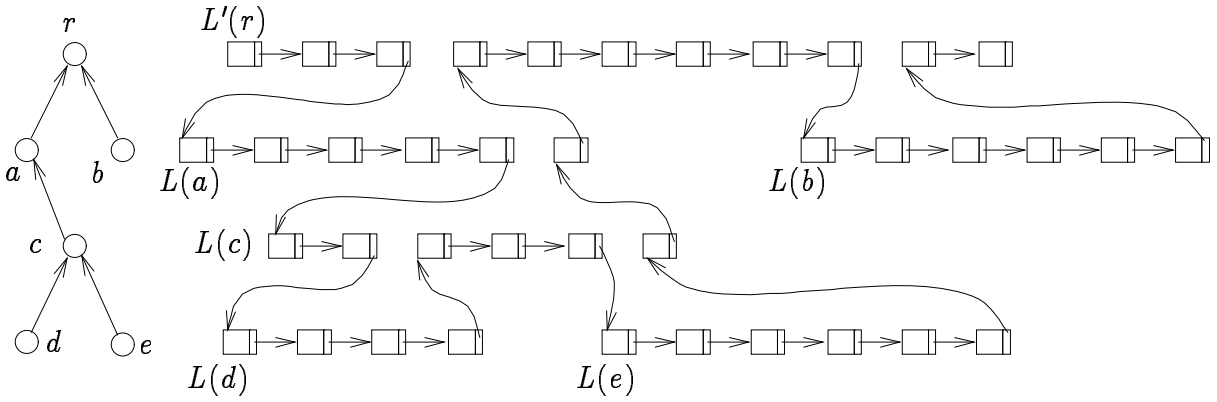

Figure 4: The effect of the plugging step execution by all vertices of a tree but the root r. On the right is $L'(r)$ after the execution of the plugging step.

plugged is after the twin edge (w, v) contained in $L(w)$ (Figure 3, bottom). This ensures exclusive writing.

Note that the effect of having all $v \in C - \{r\}$ perform the plugging step simultaneously is to place all the edges in their edge-lists into r's updated edge-list $L'(r)$ (Figure 4). This step takes constant time.

Growth-Control. Following the growth-control schedule of [JM91] we define as *critical size* the quantity $B = 2^{\sqrt{\log n}}$. Therefore $\log B = \sqrt{\log n}$. As we mentioned, the algorithm is divided in phases. The purpose of each phase i is to *promote* components to phase $i + 1$, i.e. to grow the size of each component, if possible, to at least B^{i+1}. Therefore, at most $\lceil \sqrt{\log n} \rceil$ phases are needed. We require that each component C entering a phase i has an edge-list $L(C)$.

In light of the above discussion, if we are able to assure that, after each phase, all components that were large enough to be promoted have really been promoted, then the MST algorithm is simply composed of the following loop:

Algorithm MST
`Main Procedure`

for $\lceil \sqrt{\log n} \rceil$ times **do**
execute procedure `phase`

We will give the description of procedure `phase` in Section 3.3.

3.2 The B-list

The running time of our algorithm depends on the following observation. Since the purpose of each phase is to grow the size of a component by a factor of B, then, during any phase, components need not keep track of all the edges in their edge lists. In particular, assuming that $L(C)$ contains no internal and no multiple edges, components need to keep track of only the "best" (i.e. least weight) B edges of their edge-list $L(C)$. (A similar observation was made also in [GGS89] for their related merging components problem.) The components will do so by placing these B edges into a new list, called B-list(C). During a phase, some of the edges in B-list(C) will be used for hooking, and some will be found to be internal, i.e. connecting vertices inside the same component. Note that, if during a phase some component finds that all the edges in its B-list are either used or internal, then the component can determine that it is promoted.

In order to be able to use the B-lists of the vertices, we must initialize our data structures appropriately.

`Procedure Initialization`

1. Form n trivial trees, one per vertex (component) v.
2. For each component $v \in V$, form its linked list $L(v)$ and its B-list(v).

To compute B-list(v) for each v in parallel, we may use a selection algorithm [Col88a, Vis87, CY85]. Using the algorithm by Cole [Col88a], we can select the $B+1$-st least weight element b in time $O(\log n)$ using almost $n/\log n$ EREW PRAM processors. Then, edges with weight less than b will be copied into B-list(v).

3.3 Description of a Phase

As we have said, the algorithm is composed of $\sqrt{\log n}$ phases. Each phase will operate on the components, and will promote them to the next phase in $O(\log n)$ time. It will also do some housekeeping to prepare the data structures for the next phase.

Each phase i is divided further into $O(\sqrt{\log n})$ sub-phases. During each sub-phase j, components will hook and merge achieving a minimum size of $B^i \cdot 2^j$ vertices. We will use the variable $counter_C(j)$ to record a lower bound of the size of the component C during the sub-phases j of the phase. Whenever $counter_C(j) \geq B$ for some sub-phase j, the component has been promoted and need not take part in the remaining sub-phases of the phase.

A high-level description of a phase follows. Subsection 3.4 examines the operations performed during a sub-phase in more detail.

`Procedure phase(i)`

1. For each component C, set $counter_C(0) \leftarrow 1$.

2. Run procedure `sub-phase(j)` for $j \leftarrow 1$ to $\lceil\sqrt{\log n}\rceil + 1$. During each sub-phase components perform hooking and merging, and grow in size. As we will describe in the following subsections, each sub-phase takes time $O(\sqrt{\log n})$, therefore this step takes time $O(\log n)$. We will show that at the end of this step we have computed a minimum spanning forest of promoted components.

3. Finish up the work that was deferred during the sub-phases. The description of step 2 of the sub-phases will clarify the need for this step. In brief, if some component that formed during the sub-phases was too large and had not enough time to clean up its data structures, it will do so in this step. At the end of this step, components are implicit rooted stars, that is, for each vertex x, $p(x)$ is the root of its component.

4. Rename edges (x, y) as $(p(x), p(y))$, where $p(x)$ is the root of x's and $p(y)$ is the root of y's component. Internal edges in the components' edge list are easily identified, since they have identical endpoints; they are given weight of $+\infty$.

5. Sort edges lexicographically according to their endpoints. We can use Cole's Mergesort algorithm [Col88b] for this purpose, which sorts m elements in $O(\log m)$ time using m EREW PRAM processors. We should remark that actually an integer-sorting or a bucket-sorting algorithm suffices for this purpose. On the sorted list, multiple edges end up in a sequence. Then, using list-ranking we find for each sequence of multiple (x, y) edges, the one with minimum weight. This edge is recorded as *useful* while the remaining multiple ones are given weight of $+\infty$.

6. Remove internal and multiple edges from the edge-lists by $O(\log n)$ pointer jumping steps. Recompute the *twin* pointers of the useful edges as follows: First, observe that, after removing redundant edges from an edge-list, all edges named (v, w), useful and redundant, point at the same location. This location is the edge (v', w') that comes lexicographically after (v, w). The useful edge (v, w) passes its address to a field $prev((v', w')$. From there, the useful edge (w, v) reads it, by following pointer $next(twin((w, v)))$.

7. For each component C, form its B-list(C) to enter the next phase $i+1$ as follows: Determine the $B + 1$-st element b in $L(C)$. Then, copy edges smaller than b into a new list, B-list(C).

We discuss now the implementation of procedure phase in the EREW PRAM model. Each step runs in $O(\log m)$ parallel time. Step 2 (the sub-phases) requires $m/\sqrt{\log m}$ processors, step 5 needs m processors, and the remaining steps use $m/\log m$ processors. Thus, the algorithm uses $O(n+m)$ processors. Assuming that there exists an integer sorting algorithm that runs in logarithmic time using $O(m/\sqrt{\log m})$ processors, the whole algorithm will have this processors bound. In fact, the algorithms given in [KRS90] and in [She91, HS90] are within the desired bounds. However, due to space requirements (the former) and to unrealistic machine assumptions (the latter), these algorithms are not considered practical.

3.4 Description of a Sub-phase

As we have said, each component C entering a phase holds B-list(C), a linked list of its B least-weight, outgoing and non-multiple (useful) edges. The idea behind the component's B-list is described in this invariant:

Invariant 1 *In the beginning of each sub-phase, the B-list of any active, unpromoted component contains enough least-weight edges to promote the component to the next phase.*

This is certainly true in the beginning of the first sub-phase of a phase because B-list(C) contains edges leading to B distinct components. During each sub-phase s, components hook and merge achieving size of at least 2^s. Assuming that some component C has collected since the beginning of the phase $k < B$ components where $k \geq counter_C(s) \geq 2^s$, C needs to keep track of only $B - counter_C(s)$ least-weight non-internal, non-multiple edges. The reason is that these edges will lead to an equal number of components that were distinct at the beginning of the first sub-phase, therefore we can assure promotion of C. We will prove that this invariant will be preserved through each sub-phase. Lemma 1 shows that we can select correctly the edges in this group and, in fact, the $B - counter_C(s)$ least-weight outgoing edges to promoted components appear in C's B-list.

Each sub-phase proceeds as follows (Figures 5 and 6):

```
Procedure Sub-phase(j)
```

1. The root v of each active component finds the best edge in its B-list, say (v,w), and moves it to the front of the list. We call this step the hooking step, since we can think implicitly of v hooking to the component of which w is a member, by creating a pointer $p(v) = w$ (Figure 5). As we have mentioned, this step implicitly creates pseudotrees with circumference two.

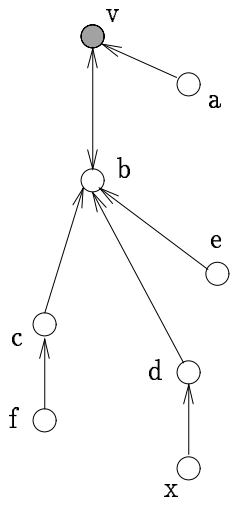

Figure 5: RUN OF A SUB-PHASE. (See also the next figure.) The implicit pseudotree with circumference two of some components that hooked together at the beginning of a sub-phase forming component C. The vertex v that will become the root of C is shown shaded.

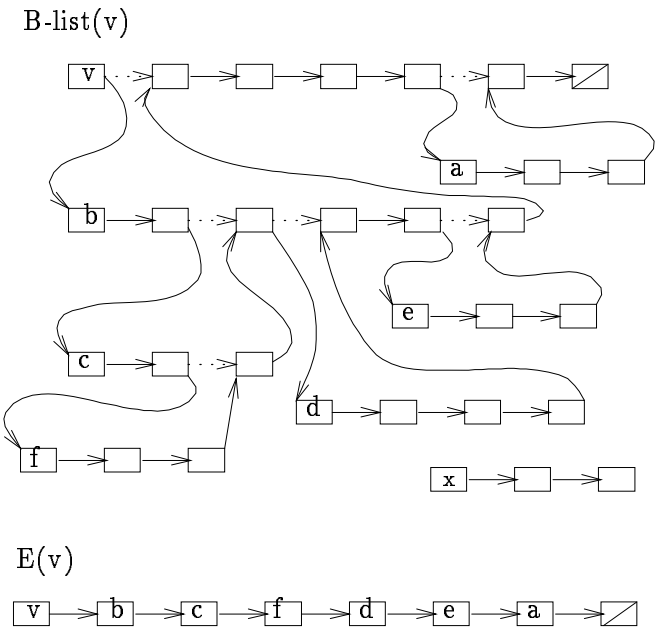

Figure 6: (Top) The effect of the edge-plugging of the B-lists. Labeled edges represent the edges that the named component used for hooking and hold the *counter*s of their components. Dotted pointers are those changed during the edge-plugging process. The edge-list marked with x, belongs to some component x which, though it belongs to C, could not plug its B-list because $(d, x) = twin(x, d)$ was not included in B-list(d). This component will not be counted in $counter_C(s)$. (Bottom) The $E(C)$ list, created by removing the 0-edges (unlabeled in the picture) from the B-lists. Running list rank on $E(C)$ we can enumerate and identify the components (all but x) that formed C. Before starting a new sub-phase, $B(C)$ is formed by including the $B - counter_C(s)$ least-weight outgoing useful edges. Note that $E(v)$ corresponds to a preorder traversal of the C's implicit tree (shown in the previous picture).

2. Components that hooked in the previous step perform edge-plugging in two parts. Let (v, w) be the hooking edge. In the first step components having $id(v) > id(w)$ perform the edge-plugging. In the second part, components having $id(v) < id(w)$ perform the edge-plugging iff vertex w did not plug its B-list into v, i.e., iff

$$next(twin(first(B(v)))) \neq first(B(v))$$

After these two steps, all but the roots of components have plugged their edges into the root's B-list (Figure 6). In the implicit graph, this step results in creating a forest of minimum spanning trees (instead of pseudotrees). Components that hooked but did not plug their B-lists during this sub-phase will be the roots $r(C)$ of their component's trees.

Note that the B-list of some vertex x may not get plugged anywhere, because the edge that x was to get plugged into was not included in the B-list of its parent. This will not affect the invariant on the B-list of the resulting component during this phase because B-list(x) contained edges with large weights; it will only underestimate the size of a component C, so the component may have size larger than $counter_C(j)$. We note that any plugging that is prevented by this condition is deferred until the end of the phase, so it is not lost. (Step 3 of procedure `phase` will take care of that.)

3. Using the plugged B-lists, we try to enumerate components of trees into $counter_{r(C)}(s)$, where $r(C)$ is the root of the newly created component C. This enumeration is done in this and the next step spending only $2\lceil \log B \rceil + 1$ time as follows: First, we make a copy of each *next* pointer into a new pointer *ptr*. Then the copy of the edge used for hooking by component C_i is assigned value $counter_{C_i}(s-1)$, and the remaining edges are assigned value 0. Using pointer jumping on *ptr* for $2\lceil \log B \rceil + 1$ steps over the 0-edges, we can "compact" each B-list, if the new component contains up to $B - 2^s$ edges. The compacted list $E(C)$ represents a preorder traversal of the implicit tree (Figure 6, bottom).

4. Run list-ranking on the computed edge-lists $E(C)$'s, and determine promotion of component C as follows:

   (a) If list ranking in some list did not terminate after $2\lceil \log B \rceil + 1$ pointer jumping steps (i.e. the *first* pointer did not reach the *last* or if the short-cutting edges encountered a 0-edge), then there were more than B^2 edges in the B-list(C). Given that each component started the phase with up to B outgoing, non-multiple edges, there are at least B components in the new component C, and therefore C is promoted.

(b) If the list ranking procedure terminated with rank in $counter_C(s)$ greater or equal to B, the component is promoted.

(c) If list-ranking terminated with rank less than B, the component may not be promoted, and it has fewer than B^2 edges in its B-list. In the remaining sub-phases of this phase, only these components will take part. We call these components, *active*. The remaining components (those recognized as promoted and those that could not plug their edges) will be given enough time at the end of the phase to finish up their pointer jumping (cf. step 3 of procedure `phase`).

5. Rename edges (x, y) according to their new endpoints as $(p(x), p(y))$.

6. Identify and remove internal and multiple edges of active components. This is done as follows: First we sort lexicographically each active component's B-list according to the edge's endpoints. Then, we use pointer jumping over internal and multiple edges for $2\lceil \log B \rceil + 1$ steps. At the end of this step, any sequence with up to B^2 internal and multiple edges are removed.

7. For each active component's B-list(C) containing more than B edges we select the $B - counter_C(s)$ least weight edges. If there are no more edges left in B-list(C), then C corresponds to a connected component of the input graph G and it is marked *done*. This component will not take part in any of the remaining phases or sub-phases.

This is the end of a sub-phase. Each step takes $O(\log B) = O(\sqrt{\log n})$ time. In terms of processors, the most expensive step is step 6 (lexicographical sort) which requires $nB = O(m)$ EREW PRAM processors The remaining steps require $O(m/\log B) = O(m/\sqrt{\log n})$ processors in the first phase, while in the remaining phases require only $O(n/\sqrt{\log n})$ processors.

4 Complexity of the algorithm

We prove the main theorem and two lemmas that are used in proving correctness and the complexity bounds.

Theorem 1 *Algorithm MST correctly computes the minimum spanning tree of a graph $G = (V, E)$ in $O(\log^{3/2} |V|)$ parallel time using $|V| + |E|$ EREW PRAM processors.*

Proof. As we mentioned before, if we repeat the hooking and merging steps for a long enough period of time, the minimum spanning tree of a connected graph is

computed. We first show that our algorithm correctly computes the MST of a given connected graph, and then we prove the claimed complexity.

In the correctness part of the proof, the crucial points are to show that in the B-list of any active unpromoted component there are enough least-weight edges to promote the component (Lemma 1), and that in every hooking step the least weight edge of the whole component is selected (Lemma 2). In the complexity part of the proof the idea comes from the growth-control schedule [JM91] and the carefully chosen critical size for promotion (Lemma 3). □

Lemma 1 *Invariant 1 holds in the beginning of each sub-phase s.*

Proof. Let C be an active unpromoted component. We first show that (i) there are enough edges in B-list(C), and then that (ii) these edges are actually the least-weight edges of the whole component.

(i) In the beginning of the first sub-phase, B-list(C) contains B edges leading to B distinct components. During each sub-phase s, components hook and merge achieving size of (at least) 2^s. Assume that during sub-phase $s > 1$, C collects k components where $k \geq counter_C(s) \geq 2^s$. In order for C to achieve the promoting size, C needs to be augmented by at least $B - k$ components. The $B - counter_C(s)$ non-internal, non-multiple edges that C holds at the end of sub-phase s are clearly enough for this task.

(ii) We now show that the edges in B-list(C) are, in fact, the least-weight edges of the whole component C.

If all components that hooked together during sub-phase s started the phase with less than B edges, the lemma holds, since the selection of the least-weight outgoing edges was done on the whole set of edges. If there was at least one component which started the phase with B edges, then the root of the component in which it participates holds at most $B - k$ outgoing edges.

We will need the following definitions: Let *promotion ceiling* $pc_i(C)$ of phase i be the weight of the B-th edge in B-list(C) that was formed in the beginning of the phase, if C had at least B outgoing edges, and undefined otherwise. Let C_l, $1 \leq l \leq k$, be the components that hooked together within a phase to promote C to the next phase $i + 1$. The idea behind this definition is that no edge with weight greater than $\min_l\{pc_i(C_l)\}$ will be used during the sub-phases of phase i for hooking.

We prove that every outgoing edge e in the component is dominated by some edge in the B-list(C). To see this, we will examine the three places in which edges are removed from consideration during phase i: the formation of B-lists, failure of a component to plug its B-list, and removal of internal and multiple edges.

1. If edge e was left out during the formation of the B-list(C_l) of its component C_l in the beginning of the phase, then $weight(e) > pc_i(C_l)$. Apparently, e will not be needed during this phase.

2. If edge e belonged to some component C_x which failed to plug its B-list into the B-list of its parent component C', then e will not be needed, since it has $weight(e) > pc_i(C') \geq \min_l\{pc_i(C_l)\}$.

3. If edge e was left out after the selection step of sub-phase j, then it had weight greater than any edge that will be considered during the remaining sub-phases (part (i)), so $weight(e) \geq \min_l\{pc_i(C_l)\}$.

So, every outgoing edge e in the component is dominated by some edge in the B-list(C), therefore the edges in the B-list(C) are, in fact, the least-weight edges of the whole component. □

Lemma 2 *Assume that, during the hooking step of sub-phase s (step 1), some component C picks edge (v, w) for hooking. Then (v, w) is C's least weight outgoing edge, i.e.*

$$weight(v, w) = min\{weight(x, y) | x \in C, y \in C', C \neq C'\}$$

Proof. We will prove it by induction on the number of sub-phases s. The Lemma is true for $s = 1$ as we can see by examining the steps of procedure `phase`.

To see that the lemma is true for $1 < s \leq \sqrt{\log n}$, we assume it was true at sub-phase s and we will show it true at sub-phase $s + 1$.

According to Lemma 1, at the end of step 7 of sub-phase s any component C with $|C| \geq k$, where $k = counter_{r(C)}(s)$, will be always able to select the $B - k$ least-weight outgoing to active components' non-multiple edges (assuming that there are that many outgoing edges left in the component). The proof follows if we observe that the hooking step of sub-phase $s + 1$ selects the least-cost edge of the B-list(C) and therefore of the whole component. □

Finally, the running time comes from the following

Lemma 3 *If, at the end of sub-phase s of phase i, some active component C has vertex size less than B^{i+1}, then either C corresponds to a connected component of the input graph G and is done, or there is a sub-phase $s + 1$ in the current phase during which C will hook.*

Proof. Recall that an active component at the end of the sub-phase s is an unpromoted component. If B-list(C) = $\emptyset$ then, according to Lemma 1, $L(C) = \emptyset$, thus vertices in C are not connected to any other vertex of the graph and therefore

C is a connected component of the input graph. In this case, the algorithm correctly labels it as "done".

If B-list$(C) \neq \emptyset$ then $s < \sqrt{\log n}$. To see that, without loss of generality assume that in all previous sub-phases C was hooking. (If not, consider the first phase s during which C did not hook and apply the same argument.) Since C was hooking in all the previous sub-phases, its size is now $B^i \cdot 2^s < B^{i+1} \Rightarrow 2^s < 2^{\sqrt{\log n}} \Rightarrow s < \sqrt{\log n}$ So, there is another sub-phase $s+1$ during which C will hook.

□

We also have the following corollary:

Corollary 1 *There are algorithms solving the connectivity, biconnectivity, ear decomposition, Euler tours, strong orientation and st-numbering problems of a graph $G = (V, E)$ in $O(\log^{3/2} |V|)$ parallel time using $|V| + |E|$ EREW PRAM processors.*

Proof. We can easily derive a connectivity algorithm from the MST algorithm we described, by assigning arbitrary distinct weights on the edges of the graph. In particular, we can assign $weight(e) = id(e)$ where $id(e)$ is the id of the processor assigned on edge e.

For the remaining problems, we note that the results in [MR86, TV85, Vis85, MSV86, AV84] use a connectivity algorithm as the most expensive subroutine.

5 Conclusions

We have presented a new, simple and implementable parallel algorithm for computing the minimum spanning tree (MST) of an undirected weighted graph $G = (V, E)$ of $n = |V|$ vertices and $m = |E|$ edges on an EREW PRAM, the weakest of the PRAM models, in $O(\log^{3/2} n)$ time using $n + m$ processors. Our algorithm quite naturally observes the condition that no more than one processor ever attempts to read from or write to the same memory location concurrently.

Our algorithm is faster by a factor of $\sqrt{\log |V|}$ than any deterministic algorithm previously known for any model that does not make use of concurrent writing. A major innovation is our discovery that necessary information about components can be extracted without ever explicitly shrinking the components. Component shrinking is a feature of every other parallel MST and connectivity algorithm known to us.

Our result represents a substantial improvement in the running time over the previous results for this problem, using at the same time the weakest of the PRAM models. It also implies the existence of a connectivity algorithm with the same complexity bounds for the EREW PRAM, therefore improving on previous work [JM91]. Furthermore, the number of processors used can be reduced by a factor of

$O(\sqrt{\log n})$, provided that there exists an practical integer-sorting subroutine which runs in $O(\log n)$ time using $n/\sqrt{\log n}$ EREW PRAM processors.

However, it is still an open question if there exists an $O(\log n)$ time deterministic algorithm that uses a polynomial number of EREW PRAM processors.

References

[AM91] R.J. Anderson and G.L. Miller. Deterministic parallel list ranking. *Algorithmica*, 6:859–868, 1991. Also: Proc. 3rd AWOC 1988.

[AS87] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36:1258–1263, 1987.

[AV84] M. Atallah and U. Vishkin. Finding Euler tours in parallel. *Journal of Computer and System Sciences*, 29:330–337, 1984.

[Ben80] J.L. Bentley. A parallel algorithm for constructing minimum spanning trees. *Journal of algorithms*, 1:51–59, 1980.

[CL93] K.W. Chong and T.W. Lam. Finding connected components in $O(\log n \log\log n)$ time on the EREW PRAM. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, January 1993.

[CLC82] F.Y. Chin, J. Lam, and I-N. Chen. Efficient parallel algorithms for some graph problems. *Communications of ACM*, 25(9):659–665, September 1982.

[Col88a] R. Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26:295–299, January 1988.

[Col88b] R. Cole. Parallel merge sort. *SIAM Journal of Computing*, 17(4):770–785, August 1988.

[CV86] R. Cole and U. Vishkin. Approximate and exact parallel scheduling with applications to list, tree and graph problems. In *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*, pages 478–491, 1986.

[CV88] R. Cole and U. Vishkin. Approximate parallel scheduling. Part 1: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM Journal of Computing*, 17(1):128–142, February 1988.

[CY85] R. Cole and C.K. Yap. A parallel median algorithm. *Information Processing Letters*, 20:137–139, April 1985.

[Eck77] D.M. Eckstein. Simultaneous memory accesses. Technical Report TR-79-6, Computer Science Dept, Iowa State Univ., Ames, IA, 1977.

[GGS89] H.N. Gabow, Z. Galil, and T.H. Spencer. Efficient implementation of graph algorithms using contraction. *Journal of ACM*, 36(3):540–572, July 1989.

[HCS79] D.S. Hirschberg, A.K. Chandra, and D.V. Sarwate. Computing connected components on parallel computers. *Communications of ACM*, 22(8):461–464, August 1979.

[HS90] T. Hagerup and H. Shen. Improved non-conservative sequential and parallel integer sorting. *Information Processing Letters*, 36:57–63, 1990.

[JM91] D.B. Johnson and P. Metaxas. Connected components in $O(\log^{3/2} |V|)$ parallel time for the CREW PRAM (extended abstract). In *Proc. of 32nd IEEE Symposium on the Foundations of Computer Science*, pages 688–697, October 1991.

[JM92] D.B. Johnson and P. Metaxas. A parallel algorithm for computing minimum spanning trees. In *Proc. 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 363–372, June 1992.

[KR84] S.C. Kwan and W.L. Ruzzo. Adaptive parallel algorithms for finding minimum spanning trees (extended abstract). In *International Conference on Parallel Processing*, pages 439–443. IEEE, 1984.

[KR90] R. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. *Handbook for Theoretical Computer Science*, 1:869–941, 1990.

[KRS90] C.P. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. *Algorithmica*, 5:43–64, 1990.

[Met91] P. Metaxas. *Parallel Algorithms for Graph Problems.* PhD thesis, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH, July 1991.

[MR86] G.L. Miller and V. Ramachandran. Efficient parallel ear decomposition with applications. Manuscript, 1986. An updated version appears in Chapter 7 of J.H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufman, 1993.

[MSV86] Y. Maon, B. Schieber, and U. Vishkin. Parallel ear decomposition search (EDS) and s-t numbering in graphs. *Theoretical Computer Science*, 47:277–298, 1986.

[NM82] D. Nath and S.N. Maheshwari. Parallel algorithms for the connected components and minimal spanning tree problems. *Information Processing Letters*, 14(1):7–11, March 1982.

[Pri57] R.C. Prim. Shortest connection networks and some generalizations. *Tech. Journal, Bell Labs*, 36:1389–1401, 1957.

[She91] H. Shen. *Efficient design and implementation of parallel algorithms.* PhD thesis, Department of Computer Science, Åbo Akademi, Finland, February 1991.

[SJ81] C. Savage and J. Ja'ja'. Fast, efficient parallel algorithms for graph problems. *SIAM Journal of Computing*, 10(4):682–691, November 1981.

[SV82] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[Tar83] R.E. Tarjan. *Data Structures and Network algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania 19103, 1983.

[TV85] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal of Computing*, 14(4):862–874, 1985.

[Vis83] U. Vishkin. Implementation of simultaneous memory address access in models that forbid it. *Journal of Algorithms*, 4:45–50, 1983.

[Vis85] U. Vishkin. On efficient parallel strong orientation. *Information Processing Letters*, 20:235–240, June 1985.

[Vis87] U. Vishkin. An optimal parallel algorithm for selection. *Advances in Computing Research*, 4:79–86, 1987.